An Object Model for Uniform Access to Heterogeneous Databases

Paul A. Schoening, Charlene A. Abrams and Michael G. Kahn
Division of Medical Informatics, Department of Medicine
Campus Box 8005, Washington University School of Medicine, St. Louis MO 63110

The vast amount of patient information collected and maintained by hospitals is seldom stored in a single database. Much programming effort is wasted on formulating specific queries for each of several data sources, rather than focusing attention on developing the intended functionality of the application. This problem becomes more apparent as more data sources become available. The most obvious strategy for dealing with this multiplicity of data sources, namely storing all data in a single database, is impractical for reasons such as security and administrative control. This paper describes one possible solution to managing access to several database systems within applications. Using object-oriented techniques, the solution identifies the commonality among database management systems and provides a uniform method of communication between applications and databases. We describe a C++-based implementation which embodies these concepts.

INTRODUCTION

It is no longer unusual for an application to execute on a desktop workstation and utilize data from several network sources. For example, a clinical decision-support program concerned with identifying potential adverse drug reactions requires access to patient-specific laboratory, pharmacy and other ancillary clinical data sources which frequently reside on different database management systems (DBMS) [1]. Software vendors have been providing powerful tools for making network communication transparent to the programmer, but the tools they provide are closely tied to a particular DBMS. To access a relational database, a CODASYL database, and a flat file database, a program would require three very different application programming interfaces (APIs).

The task of learning to use each API falls on the programmer who then must concentrate more on the details of data retrieval than on the issues of data utilization. Furthermore, as each individual database system may return data in its own unique format, the programmer is forced to convert the returned data into common types. To compound the problem, the programmer must manage the complete database interaction: initiating the network connection, verifying the connection, ensuring that data are retrieved from the correct database, watching for errors, and closing the connection. It is apparent that a uniform interface to these diverse DBMSs would reduce complexity and permit developers to focus on more important tasks.

We describe such an interface which initially was designed and implemented in LISP as part of a diabetes data-analysis program [2] and further refined and reimplemented in C++ as part of an infection control surveillance expert system at Washington University School of Medicine [3]. The issues of heterogeneous, distributed database access became important because the data for this application were contained in both UNIX Sybase and MS-DOS Paradox databases. To provide a straightforward, consistent, and portable interface to both database management systems, we developed an object model with well-defined capabilities to represent uniform access to heterogeneous databases.

UNIFORM DATABASE ACCESS

Our model of a generic database management system was based on the relational data model. The following requirements were considered necessary elements of the implementation:

- There must be uniform access to the data returned from any database. All tabular data must look the same irrespective of which database system contained the data element.
- All data must be converted into a set of well-defined types. For example, date/time data, which is stored differently by various database systems, must be converted to a common type.

0195-4210/92/$5.00

- Uniform error codes must be returned.
- A well-defined set of operations must be defined for all database management systems.

Analysis revealed that an object-oriented implementation with two base classes, Server and Database, each with a limited number of methods, would suffice to represent a database system (Figure 1). The Server maintains the connection to the physical database. It receives messages from the Database object in the form of queries and returns results in the form of tables or status indicators, ensuring that the connection is to the appropriate database. Multiple Databases may share a single Server as shown in Figure 2. An application uses a Server object as the communication gateway, but directs messages only to Database objects.

Encapsulating the interface functionality within the Database class permits state information such as database name to be retained so that the application developer can send a query without being concerned about maintaining the database environment. When a database query is made, the Database object verifies the Server connection, establishes itself as the Server's current Database, and captures any resulting error codes. Similarly, connection information must be retained by the Server so that the Database object can send queries to the Server without being concerned about the status of the connection. If communication is lost, the Server object can intercept any further attempts at communication until the connection is reestablished.

This simple two-class model provides the basic functionality for managing a database connection. It has several limitations, however. There is no support for distributed queries. Such a capability would require a true distributed DBMS since effective query processing requires access to metadata and is much more efficiently done within the DBMS itself. As with query processing, joins and views are handled best by the DBMS. For these reasons, abstraction is done only at the database level, not the relational table level. The Database/Server object model represents a server connection manager rather than a generic tool which provides database capabilities such as distributed transaction management and query optimization.

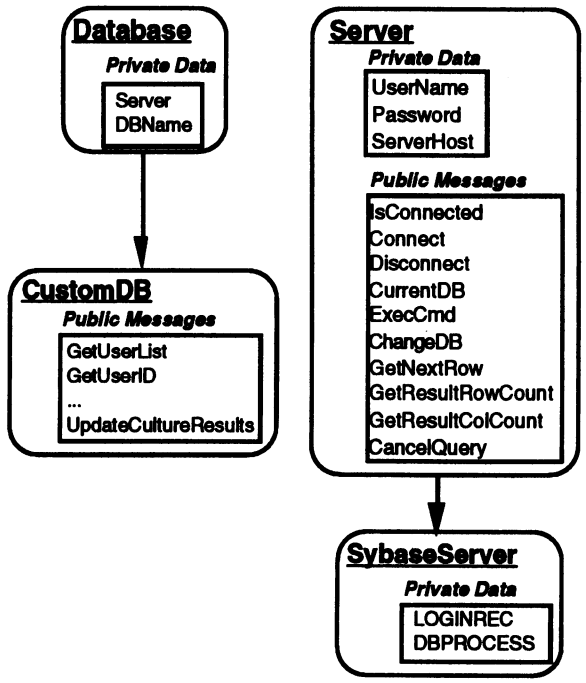

Figure 1: Database/Server Class Hierarchy. Arrows point to subclasses. SybaseServer subclass adds only private data; CustomDB subclass adds only additional messages.

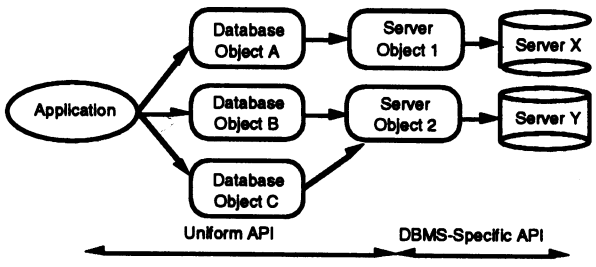

Figure 2: Application message passing through Database and Server objects.

IMPLEMENTATION FEATURES

The Server and Database classes encapsulate elements common to all database systems, but lack complete functionality. They are virtual classes from which more specific classes are created. During the implementation of our infection control surveillance system, it was necessary to derive classes from the base Server and Database classes to more closely model the individual database systems. One class is said to be derived from another if it maintains all of the data and functionality of the base class and adds data and operations that more closely model a specific system. For example, a SybaseServer class derived from the Server class is itself a Server, but with extra internal data to facilitate communication with a Sybase SQL server. The result of creating derived classes is a hierarchy of classes in which each derived class inherits the functionality of its parent.

New classes also were derived from the Database class. The primary difference between the base and derived Database classes is that the derived classes provided operations to perform specific queries. For example, rather than constructing a recurring query and passing it to a database object via the Database object's do_command operation, a new operation, whose responsibility was to perform only that query, was added to the derived class. With this technique, only one message need be passed to the Database object in order to execute a query.

Two benefits are gained by embedding the queries in the object system rather than in the application. First, the implementation details are hidden from the application code, which needs to know only the calling interface. Knowledge of the data model is required, but knowledge of database-specific function calls is not.

Secondly, certain details of the actual query implementation can be changed without affecting the application. For example, some databases (such as Sybase) support *stored procedures*, which are programs residing in the physical database. Sybase queries are performed either directly, by executing an SQL command, or indirectly, by invoking a stored procedure containing the query. Consequently, a query can be altered or moved to a stored procedure without affecting any other aspects of the program (Figure 3). We have exploited both benefits on fully-deployed, mission-critical clinical decision-support systems.

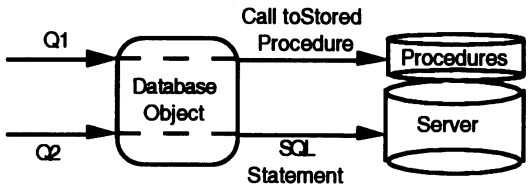

Figure 3: Query Q1 implemented as a stored procedure; query Q2 implemented as a passed SQL statement. The method of query execution is hidden and can be changed without modifying the application.

To make the Server and Database classes as easy as possible to use, other transparent features were included. For instance, managing the process of establishing a network connection and disconnecting it in a timely fashion were hidden from the user. Whenever a new Server object is created, the network connection is automatically established. When the program determines that the connection is not needed because its associated Server is no longer accessible to the program, the Server connection is dropped. This keeps the programmer from unintentionally leaving database sessions connected after the program terminates. Similarly, Database objects check that their Server is available when created and guarantee that the Server's current database is the intended one.

Several additional abstract data types (ADTs) have been created to assist in manipulating the data returned from database queries. Each ADT is implemented as a C++ class. Since all data returned from relational database systems are tables, the primary data type is the Table. A Table can hold any number of rows and columns with varying types of data in the columns. Determination of column types is made as the table is created, using the data types returned from the query. It remains the programmer's responsibility to specify a table element's type before using it in another operation. The column element type is responsible for implicit type conversion from one standard type to another, so some of the programmer's burden is reduced in this area. Each element of a table row is an instance of a ColElt ADT. Operations on ColElts provide automatic type conversion from standard C++ types to the common types established for database access. The C++ provision for overloading common operators, such as the assignment operator, permits the straight-

forward implementation and use of these implicit type conversions. Thus, operator overloading allows a complex type to be manipulated as if it was a standard type. For example, the Row and Table ADTs are accessed as simple arrays utilizing this method.

DISCUSSION

Our implementation has worked well in two infection control surveillance systems. We have been able to test the functionality of the Database/Server object classes independently of these applications and have made modifications during the course of development with little impact on the application code. Especially useful were the isolation of queries in specific class operations and the ability to return entire tables as query results without requiring any foreknowledge of the table size. Sybase returns only one row at a time, but the object system retrieves the entire table and returns it to the application. This obviates the need for the application to allocate memory for the table. The ability of the Database class to return entire tables also allows the programmer to check the row count before using the data.

Because all database operations allocate memory for the data they return, the programmer is able to keep previous query results in active memory for as long as desired without having to copy them first. When the data are no longer needed a simple *delete* allows that memory to be reclaimed for future use. For example, a table containing valid user names and IDs is retrieved and used throughout the execution of the program. The query that retrieves the data returns the entire table, which is kept until the program terminates. Further queries have no affect on this table.

There are some limitations to the class model presented in Figure 1. It may be too general to be practical for all purposes. Many applications require only character string representations of the data to be returned. Our model returns the data in its native type and it is the programmer's responsibility to convert it into other types as necessary.

The classes provide streamlined access to multiple databases, but do not permit the synthesis of data from separate, distributed sources. For example, one cannot structure a Database object query to retrieve data from two different Servers or to perform a join using tables from two different databases. It was a design decision to require only one Server per Database and modifying the model to permit more than one would vastly complicate query handling. The burden of query processing would then move from the actual DBMSs to the classes. We elected not to implement a general-purpose unified object-oriented front-end for all databases.

Other investigators have developed object-oriented models for accessing traditional database systems. Gagliardi developed the Operational Integration System (OIS), a three-level object model for accessing heterogeneous databases [4]. Richardson developed a two-level object model to addresses the semantic and syntactic gaps between applications and databases [5]. In the KOPERNIK system, Czejdo developed an object-oriented data model in Smalltalk which provides a uniform interface to underlying relational database systems [6]. FBASE by Mullen is a more ambitious attempt to define a unified object model to federated databases [7].

The capabilities of the Database/Server object model have provided significant programming support for access to multiple disparate databases. We have created an object toolkit for reducing the managerial tasks of communicating with several remote database management systems. Even though data associated with an application resides on several distributed systems of varying architectures, the task of gathering the data can be greatly eased by utilizing a common interface to each.

ACKNOWLEDGMENTS

This work is supported by NLM Grant 5-R29-LM05387, Office of Human Genome Research Grant RO1-HG00223, and NCI Contract N01-CM-97564.

Reference

[1] Blum BI. *Clinical Information Systems*. New York NY: Springer-Verlag, 1986.

[2] Kahn MG, Abrams CB, Orland MJ, et al. Intelligent computer-based interpretation and graphical presentation of self-monitored blood glucose and insulin data. Diabetes, Nutrition & Metabolism 1991; *4 (Suppl. 1)*: 99–107.

[3] Kahn MG, Steib SA, Fraser VJ, Dunagan WC. An expert system for culture-based infection control surveillance. Submitted to Symposium on Computer Applications in Medical Care.

[4] Gagliardi R, Caneve M, Oldano G. Operational approach to the integration of distributed heterogeneous environments. In: *PARABASE-90 International Conference on Databases, Parallel Architectures, and Their Applications*. New York, NY: IEEE Press, 1990:368–77.

[5] Richardson JD. Two-layered interface architecture. Comput Stand Interfaces 1991; *13*: 151–4.

[6] Czejdo BD, Taylor MC. Integration of object-oriented programming languages and database systems in KOPERNIK. Data Knowl Eng 1992; 7: 271–98.

[7] Mullen JG. FBASE: A federated objectbase system. Comput Syst Sci Eng 1992; 7: 91–9.